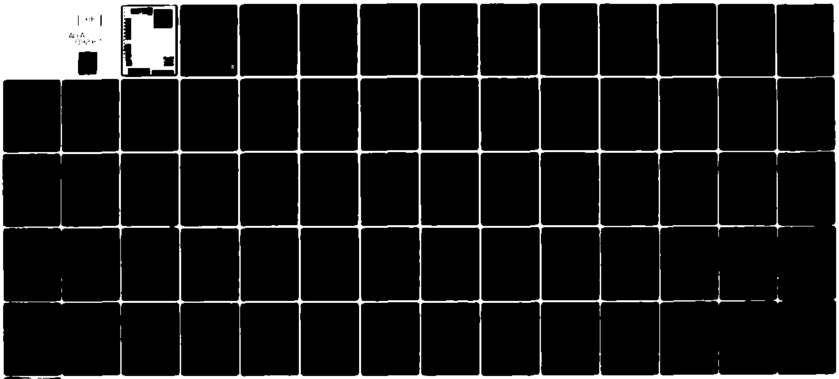
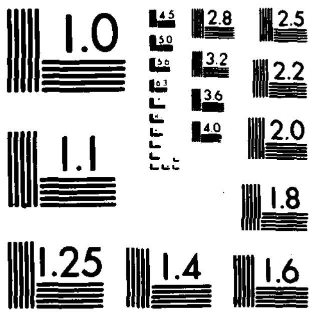
AD-A092 677 COMPUTER CORP OF AMERICA CAMBRIDGE MA F/G 9/2
MULTIBASE -- A RESEARCH PROGRAM IN HETEROGENEOUS DISTRIBUTED DB--ETC(U)
SEP 80 N00039-80-C-0402
UNCLASSIFIED NL



END
DATE FILMED
1-81
DTIC


1.0
4.5
2.8
2.5
5.0
3.2
2.2
5.6
6.3
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

LEVEL II

1

MULTIBASE -- A Research Program in Heterogeneous Distributed DBMS Technology.

Technical Report on Basic Architecture

Computer Corporation of America
575 Technology Square
Cambridge, Massachusetts 02139

September 1980.

This research was jointly supported by the Defense Advanced Research Project Agency of the Department of Defense and the Naval Electronic System Command and was monitored by the Naval Electronic System Command under Contract No. N00039-80-C-0402. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Naval Electronic System Command or the U.S. Government.

Table of Contents

1. Introduction

1.1 What is Multibase?

The database approach to data processing requires that all of the data relevant to an enterprise be stored in an integrated database. By "integrated", we mean that a single schema (i.e., database description) describes the entire database, that all accesses to the database are expressed relative to that schema, and that such accesses are processed against a single (logical) copy of the database. Unfortunately, in the real world many databases are not integrated. Often, the data relevant to an enterprise is implemented by many independent databases, each with its own schema. Such databases are nonintegrated. Furthermore, these databases may be managed by different database management systems (DBMS), perhaps on different hardware. In this case, the databases are distributed and heterogeneous, in addition to being nonintegrated. Thus, the real world of nonintegrated, heterogeneous, distributed databases differs greatly from the more ideal world of an integrated database.

Nonintegrated, heterogeneous, distributed databases arise for several reasons. First, many of these databases were created before the benefits of integrated databases were well understood. In those days, total integration was not a principal database design goal. Second, the lack of a central database administrator for some enterprises has made it difficult for independent organizations within an enterprise to produce an integrated database suitable for all of them. Third, the large size of many data processing applications has made distribution a necessity, simply to handle the volume of work. Since integrated distributed DBMSs have not been available, it has been necessary to implement applications on different machines. Since different applications often have different performance and functionality requirements, different DBMSs were often selected to run on these machines to meet these different requirements. Many data processing organizations have experienced these problems, and so there are many nonintegrated, heterogeneous, distributed databases in the world.

A principal problem in using databases of this type is that of integrated retrieval. In such databases, each independent database has its own schema, expressed in its own data model, and can be accessed only by its own retrieval language. Since different databases in general have different schemata, different data models, and different retrieval languages, many difficulties arise in formulating and implementing retrieval requests (called queries) that require data from more than one database.

These difficulties include: resolving incompatibilities between the databases such as: differences of data types and conflicting schema names; resolving inconsistencies between copies of the same information stored in different databases; and transforming a query expressed in the user's language into a set of queries expressed in the many different languages supported by the different sites. Implementing such a query usually consumes months of programming time, making it a very expensive activity. Sometimes, the necessary effort is so great that implementing the query is not feasible at all.

Multibase is a software system that helps integrate nonintegrated, heterogeneous, distributed databases. Its main goal is to present the illusion of an integrated database to users without requiring that the database be physically integrated. It accomplishes this by allowing users to view the database through a single global schema and by allowing them to access the data using a high level data manipulation language (DML). Queries posed in this language are entirely processed by Multibase as if the database were integrated, homogeneous, and non-distributed. Multibase uses the Functional Data Model to define the global schema, and the language ADAPLEX (DAPLEX imbeded in ADA) as the high level DML. The functional data model and ADAPLEX are discussed in Section 5.

1.2 Implementation Objectives

There are many approaches to the design of the Multibase system. In deciding which approach to choose, we begin with the following design objectives.

1) Generality: we do not want to design an application specific Multibase system. Instead, we want to provide powerful generalized tools which can be used to integrate various database systems for various applications with a minimum of programming effort.
2) Extendability: we want a design which allows expansion of functionality without major modification. There are areas in the Multibase design where substantial research effort is still required, and so we must be able to add additional features to the Multibase system as results come in.
3) Compatibility: we want a design that does not render existing software invalid, because this represents a very large investment. Thus, we must leave the existing interface to the local DBMS intact.

The proposed architecture of the Multibase system consists of two basic components: a schema design aid and a run-time query processing subsystem. The schema design aid provides tools to the "integrated" database designer to design the global schema and to define a mapping from the local databases to the global schema. The run-time query processing subsystem then uses the mapping definition to translate global queries into local queries, ensuring that the local queries are executed correctly and efficiently by local DBMSs. The schema design aid is discussed first.

1.3 Schema Architecture

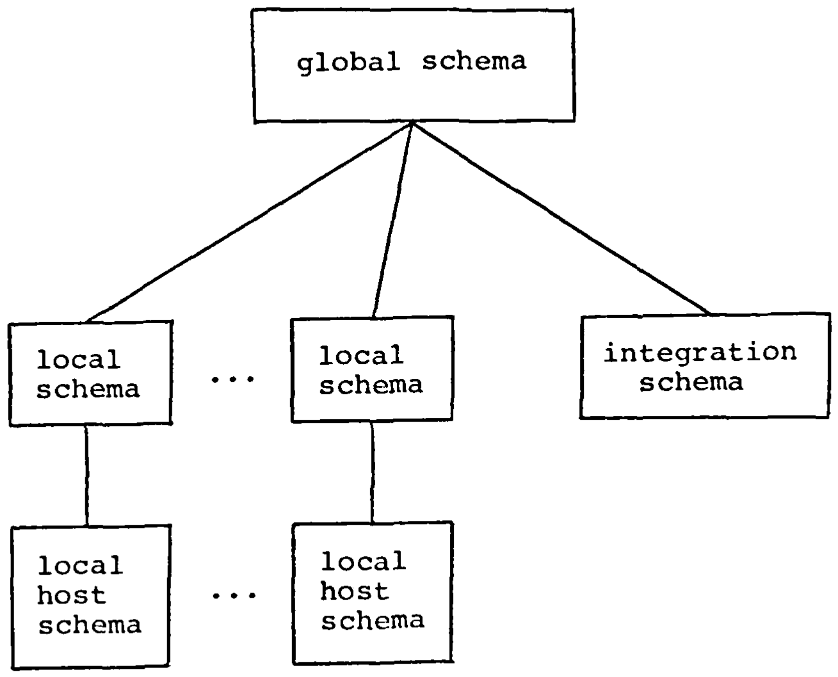
The Multibase architecture has three levels of schemata, a global schema (GS) at the top level, an integration schema (IS) and one local schema (LS) per local database at the middle level, and one local host schema (LHS) per local database at the bottom level. These components and their interrelationships are depicted in Figure 1.1.

The local host schemata are the original existing schemata defined in local data models and used by the local DBMSs. For example they can be relational, file, or Codasyl schemata. Each of these LHSs is translated into a local schema (LS) defined in the Functional Data Model. By expressing the LSs in a single data model, higher levels of the system need not be concerned with data model differences among the local DBMSs. In addition, there is an integration schema that describes a database containing information needed for integrating databases. For example, suppose one database records the speed of ships in miles per hour, while the other records it in kilometers per hour. To integrate these two databases, we need infor-

---

Schema Architecture

Figure 1.1



---

mation about the mapping between these two scales. This information is stored in the integration database.

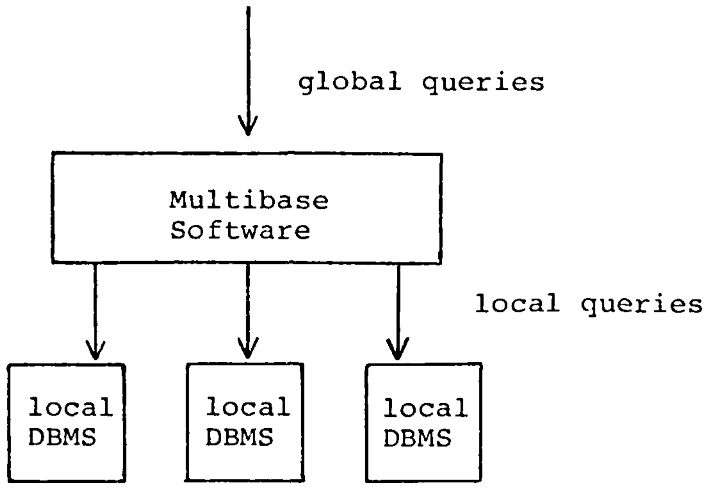
The LSs and IS are mapped, via a view mapping, into the global schema (GS). The GS allows users to pose queries that appear to be on a homogeneous and integrated database. Roughly speaking, the LHS to LS mapping provides homogeneity and the LS and IS to GS mapping provides integration. The schema design aid provides tools to the database designer to define LSs, the GS, and the mapping among them and the LHSs.

1.4 Query Processing Architecture

The architecture of the run-time query processing subsystem consists of the Multibase software and local DBMSs. These components and their interrelationships are depicted in Figure 1.2. The users submit queries over the global schema (called global queries) to the Multibase software which translates them into subqueries over local schemata (called local queries). These local queries are then sent to local DBMSs to be executed.

Since the global queries are posed against the global schema without any knowledge of the distribution of the data and the availability of "fast access paths", the Multibase software must optimize queries so they can be executed efficiently. In addition, the translation process must also be correct; that is, the local queries must retrieve exactly the same information which the original global query requests.

Run-Time Query Processing Subsystem Figure 1.2



1.5 Meeting the Objectives

The proposed architecture meets the objective of generality. The only component of the Multibase system which is customized for the application is the global schema and its mapping definition to the local schemata. The only component of Multibase that is customized for the local DBMSs is the interface software that allows Multibase to communicate with the heterogeneous DBMSs in a single language. These are only small components of the Multibase system. Thus, most of Multibase is neither application-specific nor DBMS-specific. Multibase also meets the objective of compatibility, because local databases are not modified, therefore, existing application programs can still access local databases through local DBMSs. And as the details of the architecture are discussed in later sections, it will become clear that the objective of extendability is also met.

1.6 Summary of The Report

.he architecture of the Multibase system is expanded in more detail in Section 2. The process of mapping each LHS to a LS and merging LSs into a GS is discussed in Section 3. Section 3 also discusses the problem of data incompatibility and inconsistency. The method by which user queries are translated into efficient local queries is discussed in Section 4. The mapping language ADAPLEX and the use of the language in defining the mappings between LSs and GS are discussed in section 5. Section 6 is a summary.

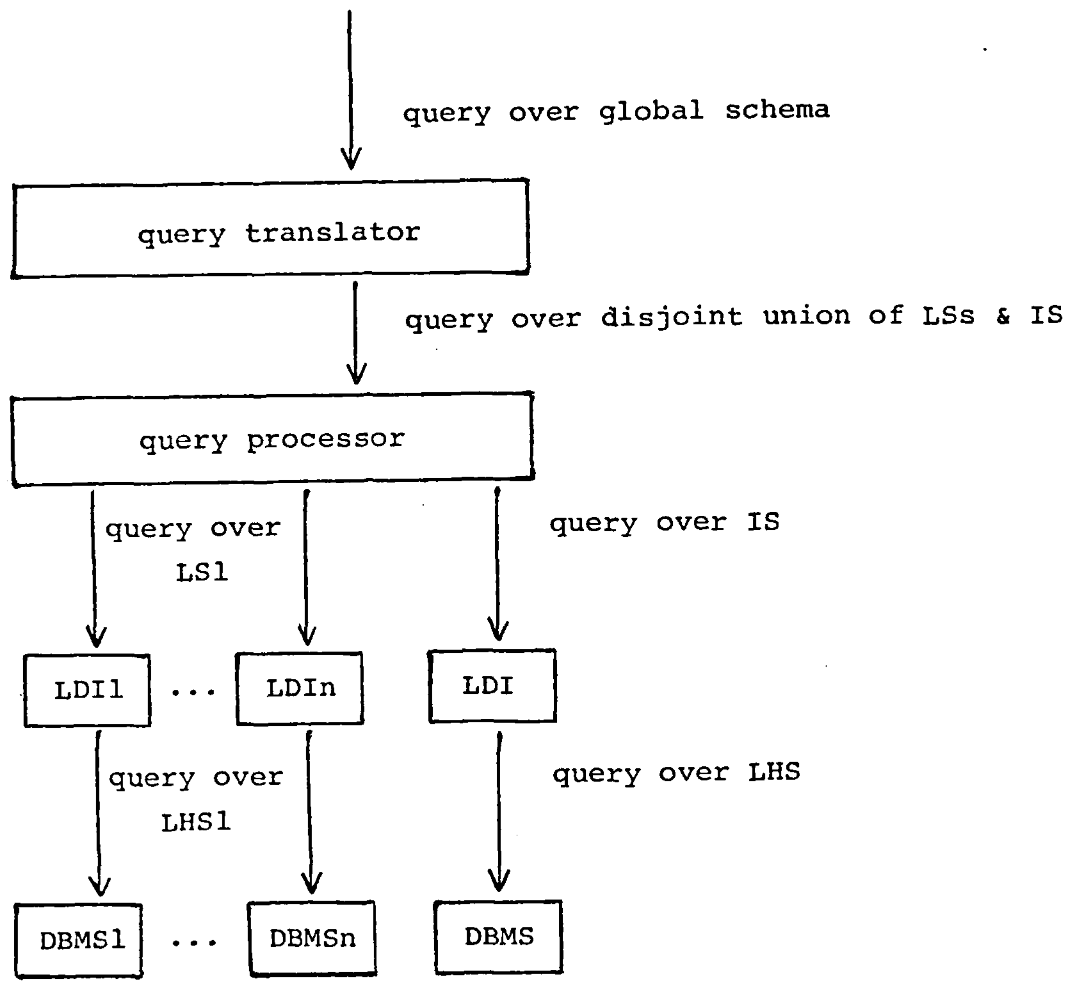
2. Run-Time Query Processing Subsystem

The architecture of the Multibase run-time subsystem consists of:

1. a query translator,
2. a query processor,
3. a local database interface (LDI) for each local DBMS,
4. local DBMSs.

A global query references entity types and functions defined in the global schema. Before it can be processed, it is first translated into a query referencing only entity types and functions defined in the local schemata, by the query translator. In other words, the query translator translates a global query over the global schema into a global query over the disjoint union of local schemata. The query processor decomposes the global query over the disjoint union of local schemata into individual local queries over local schemata. The query processor also does query optimization and coordinates the execution of local queries. The LDI translates local queries received from the query processor into queries expressed in the local DML and translates the results of the local queries into a format expected by the query processor. These components and their interrelationships are depicted in Figure 2.1.

Run-Time Query Processing Subsystem Figure 2.1



2.1 The User Interface

The global schema is expressed in the functional data model [Shipman]. In this data model, a schema is composed of <u>entity types</u> and <u>functions</u> between entity types. Each entity type contains a set of entities, so functions map entities into entities. Functions can be <u>single-valued</u> or <u>multi-valued</u>, and can be <u>partially defined</u> or <u>totally defined</u>.

The functional data model was selected because it embodies the main structures of both the flat file data models, such as the relational model, and the link structured data models, such as Codasyl. Entity types correspond roughly to relations in the relational model or record types in the Codasyl model. Functions correspond to owner-coupled sets in the Codasyl model.

The DML that we use with the functional data model is called ADAPLEX -- the data language DAPLEX embedded in the programming language ADA. DAPLEX is a high level DML that operates on data in the functional data model and is designed to be especially easy to use by end users. Computationally, it is as powerful as relational calculus. Since it is embedded in ADA, it acquires the additional power of a procedural language. Details of the language are discussed in Section 5.

2.2 Query Translator

The query translator receives global queries expressed in ADAPLEX over the GS and translates them into queries expressed in an internal language over the disjoint union of LSs and IS.

To perform the translation, the query translator must use the mapping that defines how entity types and functions of the GS are constituted from the entity types and functions of the LS and the IS. The query translator uses these mapping definitions to substitute global entity types and global functions in the global query by their mapping definitions. The substitution results in a query containing only entity types and functions of the LSs and the IS. Therefore references by the global query to entities in the GS are now expressed as references to the actual entities at particular sites that implement the global GS. Any extra data needed from the integration database to resolve incompatibilities among LSs is now explicitly referenced in the translated query.

The query produced by the query translator only references data in the LS and the IS. Thus, we can imagine that this query is posed against a database state that is the disjoint union of the LSs together with the IS. This disjoint union is a homogeneous and centralized view of the distributed heterogeneous database.

The language used for defining the mapping between schemata must be compatible with the global DML. Otherwise, it would be awkward to translate the query from the GS to LSs and IS using conventional query modification techniques (Query modification composes the given query, which is a function from GS states to answer states, with the mapping from LS and IS states to GS states, to produce a query from LS and IS states to answer states; (cf. [Ston 75].) Therefore, we propose to use the same language ADAPLEX as both the query and mapping language. The process of constructing the global schema from the local schemata is discussed in Section 3.

2.3 Query Processor

The query processor translates a query over the disjoint union of LSs and IS into a query processing strategy. This strategy includes: a set of queries each of which is posed against exactly one LS or the IS; a set of "move" operations to ship the results of these queries between the local DBMSs and the query processor; and a set of queries that is executed locally by the query processor to integrate the results of the LS and IS queries. The main goal of this translation is to minimize the total cost of evaluating the query, where cost is measured by local processing time and communication volume.

A query processing strategy is produced in two steps. First, the query is translated into an internal representation called a query graph. Using this representation, the query processor isolates those subqueries of the given query (which are essentially subgraphs of the query graph) that can be entirely evaluated at one local DBMS. So, the result of the first step is the set of single-site subqueries of the given query.

The second step is to combine the single-site queries with move operations and local queries issued by the query processor. Move operations serve two purposes. First, they are used to gather the results of the single-site queries back to the query processor. These results can be integrated by the query processor by executing a query local to itself. The integrated results may be the answer to the query, in which case they are returned to the user. Second, they may be used as input to other single-site queries. In this case, a move operation is issued to ship the data to the local DBMS that needs it. The method by which single-site queries, move operations, and queries local to the query processor are sequenced to produce a correct and efficient strategy is discussed in Section 4.

2.4 Local Database Interface (LDI)

Local queries posed against the LSs are sent by the query processor to the LDIs in an internal format. The LDI translates these local queries into programs in the local DML and programming language over the local host schema (LHS). This translation is optimized to minimize the processing time of the translated query. When the local DBMS uses a high level (i.e., set-at-a-time) language, such as DAPLEX, this translation is fairly direct. However, when the local DBMS uses a low level (i.e., record-at-a-time) language, such as Codasyl DML embedded in COBOL, this translation may be quite complex and may require nontrivial optimization. Translation methods for a file system and Codasyl language are described in Section 4.

To do the translation, the LDI must have information about how entity types and functions in the LS are mapped to objects in the LHS. These mappings are defined using the rules discussed in Section 3.1.

2.5 Initial Implementation

In this section, we have sketched the basic architecture of Multibase. Details of individual components appear in later sections. These sections outline preliminary solutions to most of the technical problems that need to be solved to build Multibase. These solutions will be the basis of a "breadboard" implementation to be completed by the end of 1980.

We emphasize, however, that Multibase is a three year research project of which only six months have elapsed. As new and better solutions to technical problems are discovered, they will be incorporated into later versions of the system. In particular, optimization (both local and global) of query processing and techniques for handling data incompatibility will be the subject of future research.

3. Schema Integration Architecture

"Schema Integration" is the process of defining a global schema and its mapping from the existing local schemata. The general architecture of this design process is discussed in this section.

There is one local host schema (LHS) for each local database. Each LHS can be a relational, a Codasyl, or a file system. To merge these LHSs we must convert them into a common data model first. Otherwise, we would be mixing relations from a relational model with record types and set types from a Codasyl model. Thus the first step of schema integration is to translate LHSs into Local Schemata (LS) defined in the Functional Data Model of ADAPLEX.

The second step is to merge LSs into a GS. To do this, an integration schema which defines an integration database is often needed. An integration database contains: information about mapping between different scales used by different LSs for the same entity type; statistical information about imprecise data; and other information needed for reconciling inconsistency between copies of the same data stored in different databases. The integration schema and LSs are then used to define a global schema.

The overall architecture of schema integration consists of:

a) a global schema,
b) a mapping language,
c) local schemata (LS) and an integration schema (IS),
d) a mechanized local-to-host schema translator,
e) local host schemata (LHS) and local DBMSs.

These components and their interrelationships are depicted in figure 3.1. The local host schemata are translated into local schemata by the mechanized local host schema translator, and local schemata and the IS are mapped into the GS by using the mapping language facility.

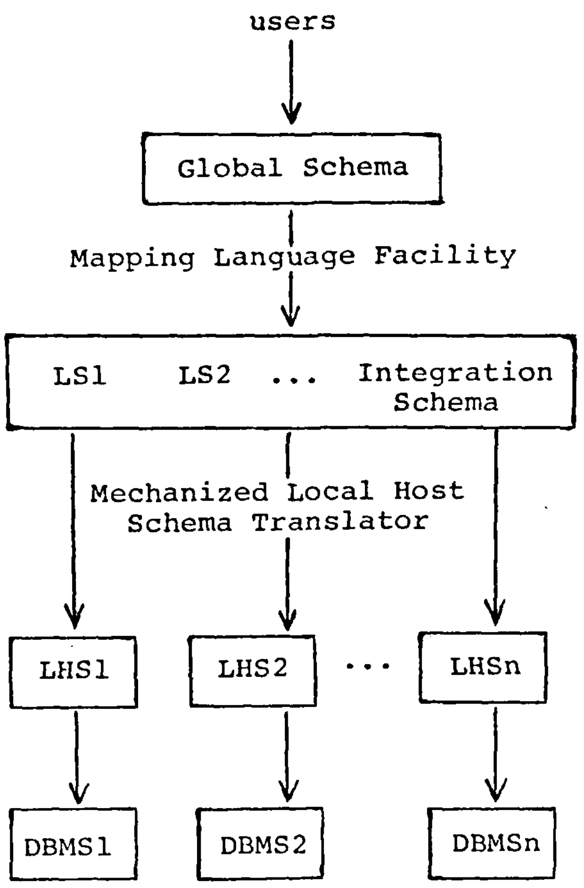
3.1 Mapping Between LHS and LS

Since a LHS can be defined in the relational, Codasyl, or file model, how a LHS is mapped into a LS depends on the data model used.

3.1.1 Codasyl Model

If a LHS is defined in the Codasyl model, then it consists of record types and set types. The functional data model consists of entity types and functions on entity types. So, to map the LHS into a LS one simply maps record types and set types into entity types and functions respectively.

Schema Integration Architecture   Figure 3.1



The concept of record type in the Codasyl model is very similar to that of entity type in the functional data model. A record in the Codasyl model has a record ID, and one or several attributes. The record ID uniquely identifies the record, and the attributes describe properties of the record. Similarly, in the functional data model, an entity is an object of interest, and the functions defined on the entity return values which describe the properties of the entity. Therefore, a record type corresponds to an entity type, and the attributes of the record type correspond to functions defined on the entity type.

If an attribute of a record type is a key (in Codasyl terminology, a key is the data item(s) declared "NO DUPLICATE ALLOWED") then the corresponding function must be a totally defined one-to-one mapping. If the attribute is a repeating group (declared to have multiple occurrences in a Codasyl model), then the function is a set-valued function.

A set type in the Codasyl model is a mapping between an owner record type and one or several member record types. A set type maps an owner record to a set of member records, or, conversely, a set type maps a member record to a unique owner record. Therefore, a set type resembles a function which maps an owner entity to a set of member entities, or, conversely, maps a member entity to a unique owner entity.

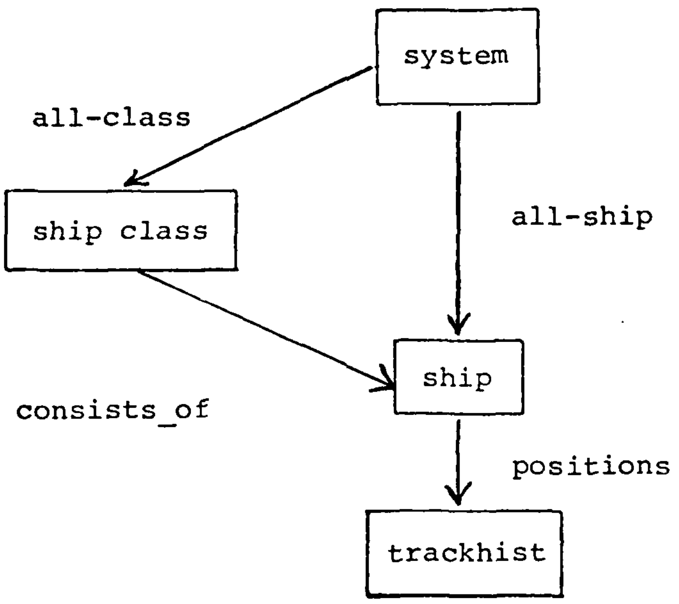
In a Codasyl model, a set type implies not only certain semantic information but also the existence of access paths. For example a set type "work-in" between "department" and "employee" record types implies that the employees owned by a department work in that department. But it also implies that there is an access path from a department record to the employee records owned by that department and another access path from each employee record to its own department record. Since the LSs will be used for query optimization, we must capture all this access path information in the LSs. Therefore, for each set type in a LHS, not only a set-valued function from the owner entity type to the member entity type, but also a single-valued function from each of the member entity types to the owner entity type must be defined in the corresponding LS.

In a Codasyl model, a record type can be declared to have a "LOCATION MODE CALC USING KEY". This means that an index file is created for the key, and the record type is directly accessible through the indexed key. Therefore, for each record type with "CALC KEY" in the LHS, a system set function of which the domain is the key value and the range is the entity type (corresponding to the record type) must be defined in the LS. This system set function will be used only for query processing optimization. It is not visible to the database designer. Therefore, it can not be incorporated into the global schema. This restriction is imposed to preserve the data independence of the global schema.

For example, the Codasyl schema shown in Figure 3.2 is translated into the schema in the functional data model shown in Figure 3.3. In Figure 3.3, the inverse of a function F is denoted by "F-inv".

A Codasyl Schema

Figure 3.2



Shipclass Record

*	classname	char(24)
	length	char(6)
	draft	char(2)
	beam	char(3)
	displacement	char(5)
	endurance	char(3)

Trackhist Record

**	DTG	char(10)
	speed	char(3)
	latitude	char(5)
	longitude	char(6)
	course	char(3)

* primary key
** key within a set

Ship Record

*	UIC	char(6)
*	VCN	char(5)
	name	char(26)
	type	char(4)
	flag	char(2)
	owner	char(2)
	hull	char(4)

A Schema in the Functional Data Model   Figure 3.3

```
type shipclass is entity           type system is entity
   classname    : string(1..24);     all-class :  shipclass;
   length       : string(1..6);      all-ship  :  ship;
   draft        : string(1..2);    end entity;
   beam         : string(1..3);
   displacement : string(1..5);
   endurance    : string(1..3);
   consists_of  : set of ship;
end entity;

type ship is entity             type trackhist is entity
   UIC    : string(1..6);          DTG        : string  (1..10);
   VCN    : string(1..5);          speed      : string  (1..3 );
   name   : string(1..26);         latitude  : string (1..5);
   type   : string(1..4);          longitude : string (1..6);
   flag   : string(1..2);          course    : string (1..3);
   owner  : string(1..2);          positions_inv : ship;
   hull   : string(1..4);          end entity;
   positions :  set of trackhist;
   consists_of_inv : shipclass;
end entity;
```

3.1.2 Relational Model

A relational database schema consists of a set of relation definitions. To translate a relational LHS to a functional LS we essentially map each relation to an entity type. A tuple of a relation in a relational model is similar to an entity in a functional data model. A tuple is uniquely identified by its primary key and has one or more attributes, just as an entity has one or more functional values. Therefore, to map a relational model LHS into a functional data model LS, for each relation in the LHS an entity type is defined in the LS, and for each attribute of the relation a function is defined on the corresponding entity type. The range of the function is the domain of the attribute. If the attribute is a primary key, then the function must be totally defined and one-to-one. If it is a candidate key, then the function can be partially defined, but it must still be one-to-one. In any case, due to the relational format, the function must be single-valued, not set-valued. For example, the relational LHS shown in figure 3.4 is translated into the functional data model LS shown in figure 3.5.

A Relational Model — Figure 3.4

Relation Platform

	Vessel Name	char(26)
	class	char(25)
	type	char(6)
	hull	char(6)
	flag	char(2)
	category	char(4)
	PIF	char(4)
*	NOSICID	char(8)
	IRCS	char(8)

Relation Position

*	PIF	char(4)
	NOSICID	char(8)
	DTG	char(10)
	latitude	char(5)
	longitude	char(6)
	bearing	char(3)
	course	char(3)
	speed	char(3)

* primary key

A Schema in Functional Data Model Figure 3.5

```
type platform is entity
   Vessel Name   :string (1..26);
   class         :string (1..25);
   type          :string (1..6);
   hull          :string (1..6);
   flag          :string (1..2);
   category      :string (1..4);
   PIF           :string (1..4);
   NOSICID       :string (1..8);
   IRCS          :string (1..8);
end entity;

type position is entity
   PIF           :string (1..4);
   NOSICID       :string (1..8);
   DTG           :string (1..10);
   latitude      :string (1..5);
   longitude     :string (1..6);
   bearing       :string (1..3);
   course        :string (1..3);
   speed         :string (1..3);
end entity;
```

3.1.3 File Model

A key-file model consists of record files and indexed fields (keys) in those files. A record file consists of a set of records of the same type, which is similar to the concept of record type in the Codasyl model or a relation in the relational model. To map a key-file LHS to a functional data model LS, for each record file in LHS a corresponding entity type must be defined in the LS, and for each field of the record file a function must be defined on the entity type. Since a key supports an access path to the record file, for each key of a record file, a system function must be defined whose domain is the key field's entity type and whose range is the entity type corresponding to the record file. This system function is not visible to the database designer; it is used only for query optimization.

3.2 Integration of LSs

To integrate LSs into a global schema, the database designer designs an integration schema which defines an integration database. He then designs a global schema and defines it in terms of the LSs and the Integration Schema by using the view support facility.

An integration database contains information needed for merging entity types and their functions. For example, two entity types, E1 and E2, from two schemata are shown in figure 3.6. These two entity types represent information about ships. There are two functions defined on each entity type; one function returns the ship-id of a ship and the other returns the ship-class of the ship. The ship-class of E1 and E2 are coded differently. A sample of entities and their functional values are also shown in figure 3.6. To merge E1 and E2 into a single entity type, a uniform code must be defined and the two existing codes must be mapped to the new code. Definitions of the new code and the mapping function are shown in figure 3.7, and a sample of the function is shown in figure 3.8. The definitions of the new code and the function are stored in the integration database. A global schema defined on the two local schemata and the integration schema is shown in Figure 3.9.

Local Schemata

Figure 3.6

```
type E1 is entity                    type E2 is entity
    shipid1 : integer;                   shipid2 : integer;
    class1  : code1;                     class2  : code2;
end entity;                          end entity;
```

E1	shipid1	class1
e11	1212	c1
e12	1240	c3
e13	2341	c5

E2	shipid2	class2
e21	3440	d2
e22	3651	d3
e23	4411	d4

Integration Database

Figure 3.7

```
type code is entity
end entity;

Define a new function
    f : (code1 union code 2)  -> code.
```

Sample of Function f

Figure 3.8

Sample of function f

code1,code2	c1	c2	c3	c4	c5	d1	d2	d3	d4
code	1	2	3	4	5	6	7	8	9

Global Schema and The Mapping ᵣigure 3.9

```
type E is entity
     shipid : integer;
     class  : code;
end entity;
```

As the discussion above indicates, integration of local schemata which are not disjoint involves two activities: merging of entity types and merging of their functions. These activities are discussed in Sections 3.3. Two special problems relating to schema integration, the creation of new entity types, and the integration of incompatible data, are discussed in Sections 3.4 and 3.5 respectively.

3.3 Merging Entity Types and Functions

To merge two entity types, say E1 and E2 in Figure 3.6, into an entity type, say E in Figure 3.9, the database designer must first determine whether the set of entities of type E1 is disjoint from the set of entities of type E2. If E1 and E2 are disjoint, then E is simply the union of E1 and E2. If E1 and E2 are not disjoint, then the condition under which two entities from E1 and E2 respectively are identical must be specified. To specify the condition under which entities are identical, entities of E1 and E2 must be able to be identified by their attributes. Therefore, for each entity type to be merged, a function or combination of functions of the entity type must be a primary key. Two entities from two entity types being merged can then be specified to be identical if and only if they have identical primary key values.

In Figure 3.10, entity types E1 and E2 (which are assumed to overlap), are merged into an entity type E. The syntax used is a subset of ADAPLEX. Notice that "shipid1" and "shipid2" are assumed to be primary keys of E1 and E2 respectively. Further, it is assumed that an E1 entity and an E2 entity are identical if and only if they have the same primary key values.

The Mapping Definition of Entity Type E   Figure 3.10

```
     type E is entity
        shipid : integer;
        class  : code;
     end entity;

 for each  x in E1 where not (shipid1(x) isin
                                    shipid2(E2))
 loop
 create new E(shipid => shipid1(x)
                   class  => f (class1(x)));
 end loop;
for each x in E2
 loop
 create new E(shipid => shipid2(x),
                   class  => f(class2(x)));

 end loop;
```

3.4 Creation of a New Entity Type and Its Functions

Merging two entity types into a single entity type is a special case of creating a new entity type. Essentially, a new entity type may be created which is a combination of the existing entity types. However, this combination does not create new objects in the database. Rather, it simply presents many existing objects of different types as objects of a single type to the global schema users. Properties of the new global entities are simply those that previously existed in the local schemata.

However, in some cases, a database designer may want to design a more sophisticated global schema in which new (virtual) objects derive their properties (attributes) from many dissimilar existing objects. An example is used to illustrate this process, and general principles can be drawn from the example.

Suppose a global schema with two entity types, "supplier" and "parts", is to be designed from two local schemata shown in Figure 3.11. The global schema must capture all the information contained in both schemata. Notice that in the second schema, "supplier" and "parts" entities do not exist, but their existence is implied by the presence of supplier numbers and part numbers : "sno" and "pno". To capture this information, virtual "supplier" and "parts" entities corresponding to those "sno" and "pno" must be created in the global schema. A definition of the global schema is shown in Figure 3.12. Notice that in the definition primary keys "supplier.no" and "parts.no" are used to map the new entities to existing entities in the first schema and the implied entities in the second schema.

3.5 Data Incompatibility

Several sources of data imcompatibility are discussed in this section. The objective of the discussion is to show how the proposed architecture allows us to incorporate our present understanding of incompatible data into the Multibase. The details of solutions to the problem are to be fully investigated later in the project.

Some sources of data imprecision are:

a. Scale difference.
   For example, in one database four values (cold, cool, warm, hot) are used to classify climates of cities, while in another database the average temperatures in Farenheit may be recorded.

b. Level of Abstraction.
   For example, in one database "labor cost" and "material cost" may be recorded separately, while in another they are combined into "total cost". Another

example is recording an employee's "average salary" instead of his or her "salary history" for the previous five years.

c. Inconsistency Among Copies of The Same Information. Certain information about an entity may appear in several databases, and the values may be different due to timing, errors, obsolescence, etc.

There are many other sources of data incompatibility. Data incompatibility must be resolved if different databases are to be integrated. The architecture of schema integration developed previously can be extended to handle the problem.

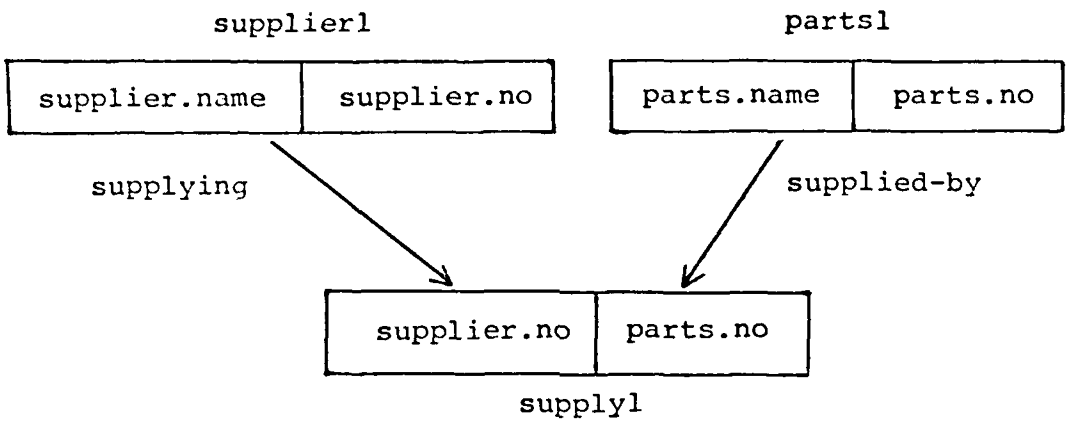
Let E1 and E2 be two entity types, and f1 and f2 be functions defined on E1 and E2 respectively. E1 and E2 have been merged into an entity type E, then f1 and f2 can be merged into the function f defined on E as follows,

$$f(e) = \begin{cases} T1(f1(e)) & \text{if } e \text{ in } E1-(E1 \text{ intersect } E2) \\ T2(f2(e)) & \text{if } e \text{ in } E2-(E1 \text{ intersect } E2) \\ g(f1(e),f2(e)) & \text{if } e \text{ in } (E1 \text{ intersect } E2) \end{cases}$$

The transformations T1 and T2 are typically used to map the ranges of f1 and f2 into a common range as discussed in section 3.3. On the other hand, the function g is used to reconcile any inconsistencies between the values of f1 and f2 over the same entity. Typically, g will involve accessing data described in the integration schema.

Two Local Schemata

Figure 3.11



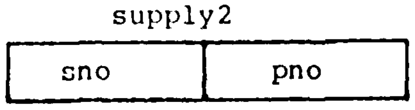
Local schema 1:

```
type supplier1 is entity
      sname       : string;
      sno         : integer;
      supplying1  : set of supply1;
 end entity;


type parts1 is entity
     pname         : string;
     pno           : integer;
     supplied-by   : set of supply1;
 end entity;


type supply1 is entity
     sno    : integer;
     pno    : integer;
 end entity;
```

Local schema 2:



```
type supply2 is entity
     sno    : integer;
     pno    : integer;
 end entity;
```

For example, in Figure 3.13, the entity types E4 and

A Global Schema Figure 3.12

```
type supplier is entity
     sno       : integer;
     supplying : set of parts;
  end entity;


type parts is entity
      name: string;
      no  : integer;
  end entity;


for each x in (sno(supplier1) Union sno(supply2))
 loop
 create supplier (sno => x);
 end loop;

for each y in (pno(parts1) Union pno(supply2))
 loop
 create parts (pno => y);
  end loop;


for each s in supplier loop
supplying(s) :+ (p in parts where(for some y1 in supply1:
      sno(s) = sno(y1) and pno(p)= pno(y1)) or
     (for some y2 in supply2 :
    sno(s) = sno(y2) and pno(p) = pno(y2));
end loop;
```

E5 are merged into the entity type E6 by using functions IS2 and IS3 of the integration database. In the Figure, the data values of the entities and functions are shown in tabular form. In this example, T1 and T2 transform the climate of cities from two different scales, (cold,cool,warm,hot) and Farenheit, into a unified scale (temperature range, probability) by combining E4 with IS2 and E5 with IS3. The function g could return all the (temperature range, probability) pairs from the two databases without any further processing, as shown in Figure

3.13, or it could use some statistical technique to process sets of (Temp range, probability) pairs, and return a simpler but descriptive summary of those pairs.

For example, the function g could return the average value and the standard deviation of the distribution represented by these pairs; it can make statistical estimation and return a confidence interval; or it can do time series analysis and return information about the spectral function. In any case, it is a research problem, and will be fully investigated later in the project.

Example of Data Incompatibility

Figure 3.13

E4 (of LS1)

city1	climate
Boston	cold
Norfork	cool
Dallas	warm
Miami	hot
...	...

IS2 (of integration database)

climate	range of temp	probability
cold	0 - 20 F	20%
cold	20 - 40 F	40%
cold	40 - 60 F	25%
cold	60 - 80 F	10%
cold	80 - 100F	5%
cool	0 - 20 F	10%
cool	20 - 40 F	20%
...	...	...

E5 (of LS2)

city2	mean temp
Denver	52 F
Chicago	54 F
Los Ang	75 F
...	...

IS3 (of integration database)

mean temp	range of temp	probability
52 F	0 - 20 F	20%
52 F	20 - 40 F	35%
52 F	40 - 60 F	30%
...	...	...

E6 (of global schema)

city	temp range	probability
Boston	0 - 20 F	20%
Boston	20 - 40 F	40%
...	...	...

4. Run-Time Query Processing Subsystem

4.1 Overall Architecture

Now we will show how the specifications developed during schema integration are utilized to drive query processing over the global schema. As we discussed in section 2, the run-time subsystem consists of a query translator and a query processor. Here we will expand these two components in further detail.

A "Global Database Manager" (GDM) is that part of the Multibase System which consists of the query translator, and the query processor. A query over the global schema is normally sent to the nearest site which has a Global Database Manager (GDM). There may be one or more GDM in a Multibase system. A GDM stores a copy of global schema, local schemata, integration schema, and the mapping definitions among them. It uses this information to parse, translate, and decompose queries over global schema into local queries over local schemata, and coordinates execution of the local queries. The structure of a GDM and its interface with local DBMSs is shown in Figure 4.1.

A query expressed in ADAPLEX over the global schema is first parsed by the parser and a parse tree is generated. Components of the parse tree, which are entities and functions of the global schema, are then replaced by their corresponding definitions which are expressed in terms of the local schemata LSs. The result is a parse tree consisting of entities and functions of the local schemata. The parser is part of the query translator.

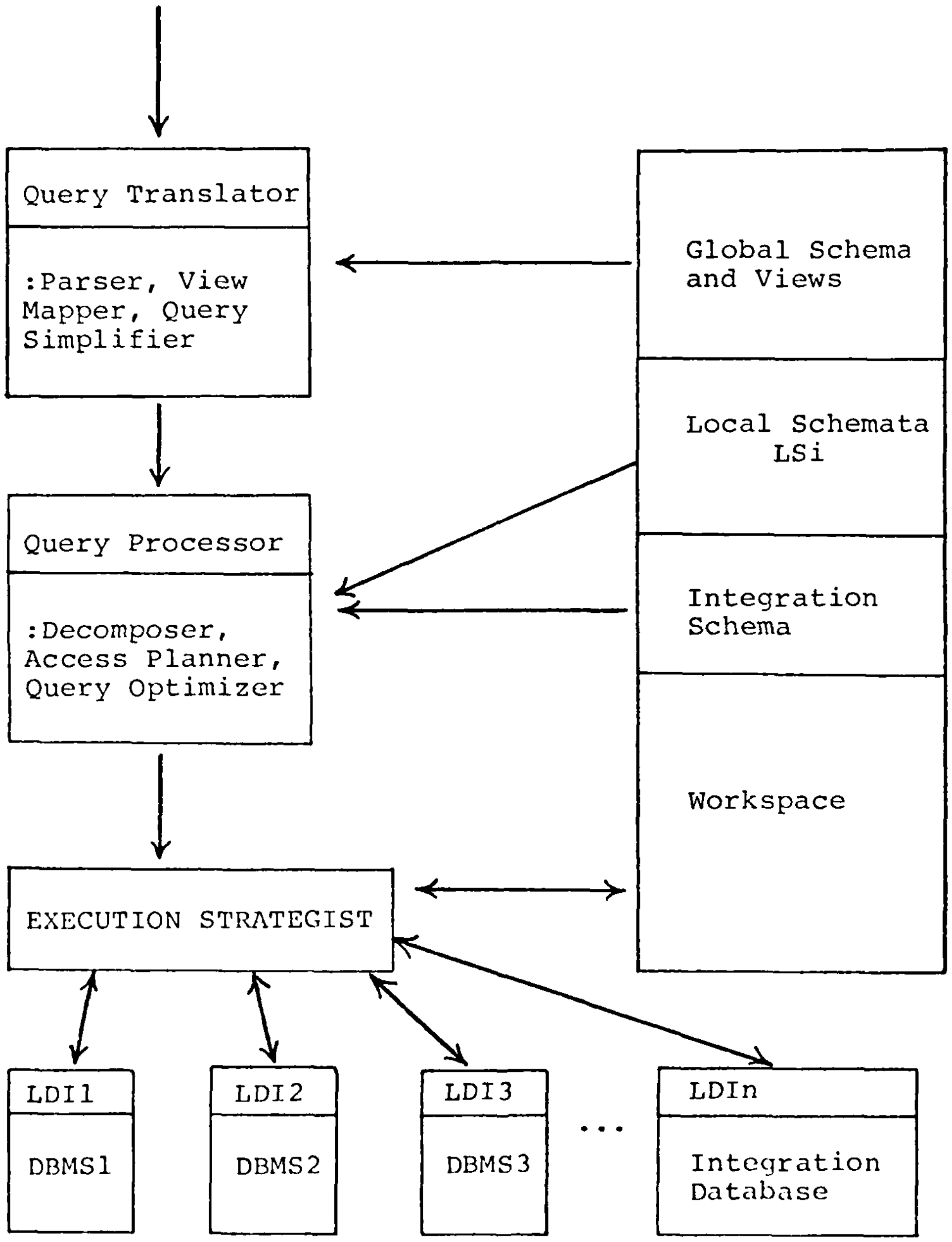
The parse tree is then simplified to eliminate the inefficient boolean components. For example, the boolean expression "(a>5)or(a<20)" is reduced to "true", and "(a>5)and(a<2)" is reduced to "false". The query simplifier is also part of the query translator.

The parse tree is then decomposed by the decomposer into subtrees. Each subtree represents a local query referencing only entities and functions of a single local schema.

The "ACCESS PLANNER" transforms the local queries into "data movement" and "local processing" steps. Depending on the memory size and processing power of each individual site, and the capacity of the communication channels, the "ACCESS PLANNER" may move data and distribute the computing load among sites, or it may move data to a central site which has large memory and computing power and do most of the processing there. In doing this planning, the "ACCESS PLANNER" tries to produce steps which minimize the cost of processing the query. The meaning of "cost" depends on the individual systems being integrated. It may mean the amount of data moved between sites, or the amount of processing time.

The execution of the access plan is coordinated by the "EXECUTION STRATEGIST". It sequences the steps of the access plan and it makes sure that the data needed by a step are there before the step is initiated.

Run Time Query Processing Subsystem Figure 4.1



The "EXECUTION STRATEGIST" communicates with local DBMSs through the Local Database Interface (LDI). The LDIs receive "data move" and "local processing" steps from the "EXECUTION STRATEGIST", translate these steps into programs in local query language or Data Manipulation Language (DML), or call local routines to process these steps, and translate the results of these steps into the format expected by the "EXECUTION STRATEGIST". The LDI may reside in a GDM if the local site does not have enough memory or cpu power; otherwise it resides with the individual local DBMS at the local site.

The query processor to be described in this section is oriented towards the initial breadboard system. It is designed to handle restricted versions of the user interface language and view mapping language with reasonable efficiency. Subsequent research is needed to extend the query processor to efficiently handle the unrestricted languages.

Within the "Query Processor", the database is modelled as a collection of _record types_ and _links_. A link L from record type R to record type S is a function from records of S to records of R; S is called the _owner_ record type and R is called the _member_ record type relative to L. We assume that if L links R to S, then L, R, and S are all stored at the same site. We also assume that there is a database schema describing the record types and links of the database.

We will sketch the Multibase query processing strategy in three steps. In Section 4.2 we define the set of queries that can be posed. In Section 4.3 we define the set of basic operations that Multibase is capable of executing. In Section 4.4 we describe how to translate a query into a sequence of basic operations that solve the query. And, in section 4.5 we describe how to translate a local query posed over a Codasyl local host schema into a program in a low level Data Manipulation Language.

4.2 Queries

A _query_ consists of a target list and a qualification. A _target list_ consists of a set of _indexed record types,_ which are terms of the form R.A where R is a record type and A is a field of R. A _qualification_ is a conjunction of selection clauses, join clauses, and link clauses. A _selection clause_ is a formula of the form (R.A op k) where R.A is an indexed record type, op is one of $\{=,\leq,<,>,\geq,\neq\}$ and k is a constant. A _join clause_ is a formula of the form (R.A = S.B) where R.A and S.B are indexed record types. A _link clause_ is a formula of the form (R-L->S) where L is a link from R to S.

Let r and s be records in R and S respectively. We say that r _satisfies the selection clause_ (R.A op k) if the A-value of r is op-related to k (written (r.A op k)).

We say that r and s _satisfy the join clause_ (R.A = S.B) if the A-value of r equals the B-value of s (i.e., r.A = s.B). And, we say that r and s _satisfy the link clause_ (R-L->S) if L connects r and s(i.e., r-L->s).

Let Rl,..., Rn be the record types referenced by qualification q, and let rl,...,rn be records in Rl,...,Rn respectively. We say that rl,...,rn _satisfy the qualification_ q if rl,...,rn satisfy all of the clauses of q.

Let Q be a query consisting of target list T=(Ril.Ajl,...,Rim.Ajm) and qualification q. Let Rl,...,Rn be the record types referenced in T and q. The _answer_ to Q is the set of all records of the form (ril.Ajl,...,rim.Ajm) such that rl,...,rn are in Rl,...,Rn (respectively) and rl,...,rn satisfy q. Given a database Rl,...,Rn and a query Q, our goal is to compute the answer to Q efficiently.

A query graph QG(N,E) is an undirected labelled graph that represents a query, Q. The nodes, N, of QG are the record types referenced in Q. Each node is labelled by the record type name of the node, the fields of the record type that appear in the target list, and the selection clauses of Q's qualification that reference the record type. The edge set E of QG contains one edge (R,S) for each join clause or link clause that references R and S. Each edge is labelled by its corresponding clause(s).

A query is called natural if (a) join clauses are of the form (R.A=S.A), that is, the fields referenced in both indexed record types in a join clause have the same name; and (b) if A is a field of two record types R and S, then R.A and S.A are "connected" by a sequence of join clauses. There is a simple and efficient algorithm that, given a database description and a query Q, renames the fields of the record types where necessary to produce an equivalent natural query Q'; Q and Q' are equivalent in the sense that they produce the same answer for any database state (up to the renaming of fields). We will therefore assume, without the loss of generality, that our queries are natural. Given that we deal only with natural queries, the edge labels corresponding to join clauses are unnecessary. Also target lists need only contain field names, instead of indexed record types.

Given a join clause (R.A=S.A) and a selection clause (R.A op k), we can deduce that (S.A op k). We assume that the qualification of each query is augmented by all clauses that can be deduced in this way. A simple and efficient transitive closure algorithm is sufficient for performing such deductions.

4.3 Basic Operations

There are three types of sites in the breadboard Multibase: OSIS, Codasyl, and GDM. Each type of site is capable of executing a different set of basic operations. This section describes these basic operations.

1. <u>OSIS Select</u>

   If record type R is stored at an OSIS site S, then the only operation that can be applied to R at S is a selection of the form:

   R[(A1=k1) and (A2=k2) and ... and (An=kn)].

   The result of the selection is a record type consisting of the set of all records r in R such that r[Ai]=ki for i=1,...,n; this result is always transmitted to GDM.

   Selections are currently implemented in OSIS, so no additional Multibase software need be written.

2. <u>OSIS Semijoin</u>

   In principle, OSIS select can be generalized into OSIS semijoin, by performing selections iteratively. Let R be an OSIS file and S a GDM file, and suppose A1,...,An are fields of R and S. Then the semijoin of R by S on A1,...,An, denoted R[A1,...,An]S, equals {r in R | (there exist s in S)(r.A1=s.A1 ... r.An=s.An)}. This can be computed by the following program.

```
Result:=0;
for each s in S
loop
   k1:=s.A1,...; kn:=s.An;
   Result:=Result U R[(A1=k1) ... (An=kn)];
end loop;
```

In practice, this operation may place an unacceptable load on the OSIS system and hence may not be usable.

3. Codasyl Tree Queries

The basic operation that can be performed at a Codasyl site S is to solve a natural tree query (defined below), returning the result to the GDM. A natural tree query Q at site S has two properties: (1) All record types referenced in Q must be stored at S. (2) Let Q' be Q minus its join clauses (i.e. all clauses of Q' are selections or links), and let QG' be the query graph of Q'; then QG' must be a tree.

To solve a tree query Q using Codasyl DML, one essentially expands the cartesian product of the record types referenced by Q and evaluates the qualification on each element of the cartesian product. We describe how this cartesian product can be systematically generated in Section 4.5.

4. Codasyl Tree Semijoins

The preceding operation can be generalized into a semijoin-like operation. Let Q be a Codasyl tree query and S a GDM record type, and suppose A1,...,An are fields of S and fields of record types of Q. Let Q' have the same qualification as Q, and the target list augmented by A1,...,An. Finally, let R' be the

result of Q'. The semijoin of Q by S on A1,...,An, denoted Q<A1,...,An], equals
{r'in R'|(there exist s in S)(r'.A2=s.A2) ... (r'.An=s.An)}.
This can be computed as follows. Suppose A1,...,An are fields of R1,...,Rn respectively where R1,...,Rn are record types of Q. (R1,...,Rn need not be distinct.) Augment the qualification of Q' by adding the clauses (R1.A1=k1) ... (Rn.An=kn). And execute the following program.

```
Result:= 0;
for each s in S loop
    k1:=s.A1;...; kn:=s.An;
    Result:= Result U Q';
end loop;
```

5. GDM Queries

The GDM can process any natural query Q provided (1) all record types referenced in Q are stored at the GDM, and (2) Q contains no link clauses. Suppose Q references record types R1,...,Rn. Q is processed by constructing a request to the local DBMS (the Datacomputer for the initial breadboard system) of the form:

```
for each r1 in R1 where (selection clauses on R1)
for each r2 in R2 where (selection clauses on R2)
                        and (join clauses on R1 and R2)
          .
          .
          .
for each rn in Rn where (selection clauses on Rn)
                   and (join clauses on R1 and Rn)
                   and (join clauses on R2 and Rn)
                    .
                    .
                    .
                   and (join clauses on Rn-1 and Rn).
        print (target list).
```

It is important that the "for" statements be in a "reasonable" order for performance reasons. Optimization techniques developed by Wong for the SDD-1 DM [Wong] are directly applicable.

4.4 Query Decomposition

To solve a query Q, we must decompose it into a sequence of basic operations. Our basic strategy is to find subqueries of Q that can be entirely solved at OSIS and Codasyl sites, move the results of these subqueries to GDM, and solve the remainder of the query at GDM.

To follow this strategy, we must isolate OSIS and Codasyl subqueries of Q. OSIS subqueries are easy to find. We simply find record types in Q that are stored at OSIS sites. For each such record type R, we produce a subquery consisting of the selection clauses on R.

Let QG be the query graph of Q. To find Codasyl subqueries, we begin by deleting from QG all record types not stored at a Codasyl site and all join clauses. Each connected component of the resulting graph includes record types and links that are stored at the same site, because no link can connect two record types stored at different sites (cf. Section 4.1). If a connected component is a tree, then it corresponds to a tree query and can be solved by the Codasyl site. If it has a cycle, then it must be further decomposed into two or more tree queries.

(In the breadboard version of Multibase, we will only handle queries whose Codasyl subqueries are tree queries; if some Codasyl subquery is cyclic, the query cannot be processed).

Having extracted the OSIS and Codasyl subqueries, we must now choose an order for these subqueries to be executed. As a first-cut solution, we propose to solve all OSIS and Codasyl subqueries before processing the results of any of these subqueries at the GDM. This strategy will be an especially poor performer if an OSIS or Codasyl subquery has no selection clauses. For such cases, we recommend use of OSIS and Codasyl semijoin operations, so that the results of some subqueries can be used to reduce the cost of other subqueries. However, this tactic brings us into the realm of new query optimization algorithms and will require further research.

4.5 Processing Codasyl Tree Queries

Let Q be a Codasyl tree query and QG its tree. The following algorithm compiles Q into a program that solves Q. The program contains statements of the form:

a) <u>for</u> R <u>in</u> set(S) ... <u>end</u> ; where S owns R via set ;

b) R:=set_inv(S) ; where R owns S via set. Note that set-inv is the inverse function of set and is always a function.

Algorithm

1. Do a pre-order traversal of QG. The result is a list of the nodes of QG. Call this list P.
2. Let R and S be nodes of QG; with R the parent of S.

   Cases

   ```
   R is the root of QG; replace 'R' by "for R" in P.
   R owns S: replace "S" by "for S in set(R)" in P.
   S owns R: replace "S" by "S=set_inv(R)" in P.
   ```

3. Push loop independent assignments up as high as possible.
4. Add an "output (target list)" statement, add selections, and joins as high as possible, tack on enough ends to balance the fors.

Example : Let QG=

$$\begin{tikzcd}
& R \arrow[dl, "L1"'] & & \\
S & & T \arrow[ul, "L2"'] \arrow[dl, "L3"'] & \\
& U & & V \arrow[ul, "L4"']
\end{tikzcd}$$

1. Preorder traversal: R,S,T,U,V.
2. P: for R

```
for S in L1(R)
    T:= L2_inv(R)
    for U in L3(T)
        V:=L4_inv(T)
```

3. T and V can be pushed up; add output statement; etc.

```
for R
    T:= L2_inv(R)
    V:= L4_inv(T)
    for S in L1(R)
        for U in L3(T)
            output (target list)
        end end end
```

5. The MULTIBASE Language

DAPLEX, embedded in the programming language ADA, is to be used both as the user-interface language and as the mapping language in MULTIBASE. In this section we first briefly review the salient features of the functional data model and DAPLEX; we then restrict DAPLEX to appropriate subsets, which will be used as the mapping language for the initial breadboard system; finally, we illustrate the use of this language for defining a global schema, and for mapping queries against a global schema into queries against the collection of local databases.

5.1 The Functional Data Model and DAPLEX

The basic constructs of the functional data model are the _entity_ and the _function_. Entities are intended to represent real-world objects, and functions to represent properties of these objects, or relationships among objects. Functions may be single-valued or multi-valued.

More precisely, in every state of the database, there is a universal set E of abstract elements. Each element in E is an _entity_ and is labelled with one or more types. Entity types are declared as follows:

```
type X is entity
end entity;
```

In any state of the database, the _extension_ of an entity type X is the set of all elements in E of type X. We denote the extension of type X also by X.

There are some pre-defined _base types_, e.g., String, Integer, Boolean, whose extensions cannot be modified. Entities of only these base types can be printed (this implies that they have _values_, which can be printed). Initially, the set E contains only these base entities. We shall describe shortly how entites of other types are created and added to E. DAPLEX permits the definition of subtypes (cf. the generalization or ISA hierarchy [Smith and Smith; Mylopoulos, et al]), e.g., _subtype_ Student _is_ Person. This states that a student entity is automatically also a Person entity; thus, the extension of student is a subset of the extension of Person.

An entity type declaration includes the declaration of all _entity functions_ applicable to entities of this type. For example:

```
type Dept is entity
            DeptNo : String;
            Emps   : set of Employee;
end entity;
```

Here we are declaring two functions; DeptNo from the set of Dept entities into the set of string entites, and Emps from the set of Dept entities into the powerset of Employee entities, i.e., DeptNo: -> String, Emps: Dept ->P(Employee).

Functions can be composed in the usual way. Thus, given major: Student -> Dept and DeptNo: Dept -> String, we can write DeptNo(major(s)) for any s in Student. Also, to compose multivalued functions (or a single-valued function and a multivalued one), DAPLEX permits the natural extension of a function f on a set X to the corresponding function (also called f) on the powerset of X. Thus,

f: X -> Y is extended to

f: P(X) -> P(Y) where, for $X' \subset X$, $f(X') = \{f(x) \mid x \text{ in } X'\}$

Similarly,

g: X -> P(Y) is extended to

g: P(X) -> P(Y) where, for $X' \subset X$, $g(X') = \bigcup_{x \text{ in } X'} g(x)$

These extended functions can then be used exactly as the original functions. Thus, we can write SocSecNo(Emps(d)), for any d in Dept.

Retrieval queries, update requests, and view definitions in DAPLEX are formulated using statements and expressions. Statements include the data definition statements, FOR loops, and the print, create, and assignment statements. (Of course, when DAPLEX is embedded in ADA, the entire armory of ADA statements can be used.) Expressions, which appear within statements, are actually set definitions, and hence may involve entity variables (ranging over entites of a specified type), functions, arithmetic comparison operations ($=, \neq$, etc.), the Boolean logical operators (AND, OR, NOT), quantifiers, set predicates (includion, containment, equality), and set operators ($\cup, \cap$, -). We shall not attempt a complete, rigorous syntactic or semantic description of DAPLEX. Rather, we give an illustrative example of schema definition, queries, and entity creation in Figure 5.1.

An entity of a specified type is created and added to the set E by means of the CREATE statement (Figure 5.1(c)). This statement permits the initialization of values to some or all of the applicable functions. These values can be updated later using assignment statements.

Figure 5.1
Example of Schema Definition, Queries, and Entity Creation

(a) schema definition

```
   type Dept is entity
            DeptName: String;
            Courses : set of Course;
            end entity;

   type Student is entity
              SNO       : String;
              Major     : Dept;
              Name      : String;
              Courses   : set of Course;
              end entity;

   type Course is entity
              CNo       : String;
              Title     : String;
              Credits   : Integer;
              end entity;
```

(b) Queries

Query1: (for each student majoring in Classics, print the student's name)

```
   for each s in Student where
       DeptName(Major(s)) = 'Classics' loop
   print (Name(s));
   end loop;
```

Query2: (for each department that has some student majoring in it and enrolled in a 6 credit course not offered by the department, print the name of the department)

```
   for each d in Dept where
   (for some s in Student : d=Major(s) and
    for some c in Course(s)-Courses(d) : Credits(c)=6)
   loop
      print (DeptName(d));
   end loop;
```

(c) Creation of a new entity:

```
   create Student (SNO => '1234', Name => 'J. Doe',
                             Major => the (d in Dept
                             where DeptName(d)='CS');
```

5.2 Subset of DAPLEX as the Mapping Language

By the "mapping language", we mean the language in which the global schema is defined as a view of the local schemata, and in which global queries are mapped into queries that can be processed against the local databases.

Theoretically, it is always possible to map a query against a view into an equivalent query against the underlying database. To see this, let D be the state of the underlying database, and V be the state of a view defined on it. We model the view definition as a function DEF from database states to view states. Let q be a query against V. We model q as a function that maps view states into states of the result. Then, there is a unique function Q (the composition of DEF and q) that makes the diagram of Fig. 5.2 commute. Furthermore, if Q is expressed as a query in any well-defined language (i.e., a language for which operational semantics can be written), then it can always be processed against the underlying database, given sufficient computational power and a workspace for storing intermediate results.

Thus, one strategy for processing global queries is by <u>view construction:</u> materialize the global state by applying the global schema definition to the local database states; then execute the global query against the global state. Clearly, this is a grossly inefficient strategy. We would like to use <u>query modification</u>

Global Query Mapping Figure 5.2

$$\begin{tikzcd}
V \arrow[r, "q"] & \text{result} \\
D \arrow[u, "DEF"] \arrow[ur, "Q"'] &
\end{tikzcd}$$

instead: use the global schema definition to modify the global query into a query against the state of the "composite schema", i.e., the disjoint union of the local databases. How easy it is to perform this syntactic transformation depends strongly upon the choice of the mapping language in which the global query and the global schema definition are expressed. Even more important, perhaps, is the problem of global optimization. The more complicated the queries after modification, the harder it will be to find efficient distributed query processing strategies.

The question, therefore, is not one of being able to process global queries at all, but rather one of practicality. It is a problem of tradeoff between convenience to users (i.e., very powerful view mechanism) and efficient implementation (i.e., fairly restricted view mechanism). Initially, we will start off with a simple, easy-to-implement view mechanism. But ultimately, we will explore the spectrum of options and select the most desirable view mechanism.

The restricted subset of DAPLEX is described in the sequel.

1. Retrieval queries:

   for each (range list) [_where_ qualification] _loop_
   {_print_ (target-list)};
   end loop;

   Here, range-list is a list of range predicates of the form "entity_variable _in_ range" where range is an entity type. Only one variable is allowed to range over each entity type. The "range-list" construct is actually a minor departure from the DAPLEX form of nested "_for each_" loops. This departure is a useful abbreviation in the context of the restricted syntax. However, the more general form of nested loops will be used as the syntax is extended.

   Qualification is a conjunction of

   i. selection clauses of the type "f(x) op c" where f is a single-valued function, x an entity variable, c a constant, and op is an arithmetic comparison operation.

   ii. join clauses of the type "f(x) = g(y)", where x and y are entity_variables, f and g are single-valued-functions that have base entities as values;

iii. link clauses of the type x = f(y) where x, y are entity_variables and f is a single-valued-function, or x isin f(y) where f is a multi-valued function;

target-list is a list of components of the type

single-valued-function := f(x) or

single-valued function := c

[here x is an entity_variable, c a constant, and f a singlevalued function.

Figure 5.3 shows how the queries of Figure 5.1 (b) can be expressed in this subset.

2. View definition:

Changing the imperative print to create results in a statement that defines the extension of a view entity type, and of those single-valued functions that take base values. For defining "links", i.e., functions that take other entities as values, the assignment statement must be used. Thus,

f(r) :+ (variable-range where qualification)

if f is multivalued;

or f(r) := the (variable-range where qualification)

if f is single-valued. Variable-range is "entity_variable in range".

Example of Queries in the Restricted DAPLEX Figure 5.3

```
(a) for each (s in Student, d in Dept)
    where d = Major(s) and DeptName(d) = 'Classics' loop
    print (Name(s));
    end loop;


(b) for each (d in Dept, s in Student, c in
    Course)
    where d = Major(s) and Credits(c) = 6 and
        c in Course(s) and not (c in Courses(d))
    loop print (DeptName(d));
    end loop;
```

Note that we preclude unnormalized structures (i.e. records with repeating groups) from the view; these must be explicity defined as hierarchies. Figure 5.4 gives an example of a view defined in this subset of DAPLEX, and shows a query on the view and, in modified form, on the underlying database.

The subset of ADAPLEX that we have just described makes the following simplifications:

1. Set expressions in range predicates and qualifications have been "flattened out", and quantifiers eliminated. This allows us to utilize existing view algorithms for relational databases. Further research will be devoted to handling the novel aspects of view processing in the DAPLEX functional model.
2. The type-subtype hierarchy is not explicitly handled. This hierarchy will no doubt be useful in the schema integration step. However, the mechanics of interpreting queries against the hierarchy require further research.

Example of A View Mapping Definition   Figure 5.4

a) View defined over the schema of Fig 5.1(a)

```
views

type enrollment is entity     type VDept is entity
   SNo        : String;              DName    : String;
   CNo        : String;           end entity;
   Credits    : Integer;
   offered_by: VDept;
   major      : VDept;
end entity;

extent

for each (s in Student, c in Course)
where c in Course(s) loop
   create enrollment (SNo => SNo (s), CNo => CNo (c),
                                 Credits => Credits (c));
end loop;

for each (d in Dept) loop
   create VDept (Dname => DeptName(d));
end loop;

for each (e in enrollment, c in Course,
        d in Dept, s in Student) loop
   offered_by(e) => the (vd in VDept where (CNo(c)=CNo(e)
        and DName(vd) = DeptName(d) and c in Course(d));
   major(e) => the (vd in VDept where d=major(s)
        and SNo(s)=SNo(e) and DName(vd)=DeptName(d));
end loop;
end extent;
```

b) Query on the view:

```
   for each (e in enrollment, vd in VDept)
   where d ≠ offered_by(e) and vd = Major(e)
      and credits(e) = 6  loop
   print (DName(vd), SNO(e));
   end loop;
```

c) Modified query on underlying entities:

```
   for each (s in Student, c in Course, d in Dept)
   where not (c in Courses(d)) and d=Major(s)
   and Credits(c) = 6 and c in Courses(s) loop
   PRINT (DeptName(d),SNo(s));
   end loop;
```

5.3 Global Schema Definition and Extension of the Mapping Language Subset

Our approach to schema integration is to provide a view support facility; the database administrator can use this facility to define the global schema as a view of the collection of local schemata. The simplest global schema is the disjoint union of the local schemata (i.e., when the view definition is the identity function); this places all of the burden of integration on the users querying the database. Alternatively the DBA can define a "more integrated" view. Providing powerful constructs (e.g., the generalization hierarchy) in the mapping language will make this task easier.

In the last section, we defined a subset of DAPLEX to be used initially as the mapping language. This subset is close in power to view mapping facilities proposed or provided in state-of-the-art DBMS. This is the subset handled by our query processing algorithm described in Section 4. We now suggest important extensions to this subset, which we feel are needed for defining the global schema as views of the local schemata in the multibase context. Extending the query processing algorithm to handle queries in this larger class will require additional research.

Schema integration often requires taking unions of entity sets from the local databases. Thus, range declarations will be extended to permit set-algebraic combinations of entity types and also functions applied to entity types. Similarly, the target list of a query or view definition statement will be extended to permit set operations.

5.4 Translating the DAPLEX Subset into Internal Form

This section relates the subset of DAPLEX described in Section 5.2 to the input language of the query processor described in Section 4.2.

Given query q in this subset of DAPLEX, transform it as follows:

1. Replace each occurrence of f(x), where x is an entity variable with range X and f a single-valued function with base values, by X.f
2. Replace every link clause of the type "x=f(y)" or "x in f(y)" by Y-f->X, where Y is the range of y and X the range of x.
3. Replace every join clause of the type "f(x)=g(y)", where f: X -> Z, g: Y -> Z, Y is the range of y, X is the range of x, and Z is not a base entity type, by X-f->Z <u>and</u> Y-g->Z.

6. Summary

This report describes the architecture of the Multibase system. Details of the components of the architecture to be implemented in the initial breadboard version are also described. Although additional research is required to fill in the details of optimization and incompatible data handling, the architecture already contains several innovative ideas in integrating distributed heterogeneous databases. These include :

i. the idea of using an integration database to resolve data incompatibility;

ii. the idea of using a mapping language to uniformly define the global schema in terms of the local schemata and the integration schema;

iii. and the idea of using query modification and query graph decomposition to transform a global query into local queries and queries over the integration database.

7. References

[Mylopoulos, et al]: Mylopoulos, J., et al, "A Language Facility for Designing Database-Intensive Applications", ACM Trans. on Database System, Vol. 2, No. 2, June 1980

[Shipman]: Shipman, D., "The Functional Data Model And the Data Language DAPLEX", SIGMOD 79, Boston, MA. 1979.

[Smith & Smith]: Smith, J., & Smith, D., "Data Base Abstractions: Aggregration and Generalization", ACM Trans. on Database System, Vol.2, No.2, June 1977

[Ston 75]: Stonebraker, M.R.: "Implementation of Integrity Constraints and Views by Query Modifications." Proc. ACM-SIGMOD Conf., San Jose, 1975. pp. 65-78.

[Wong]: Wong, E., "Retrieving Dispersed Data from SDD-1: A System for Distributed Databases", 1977 Berkeley Workshop on Distributed Data Management and Computer Network, Univ. of Cal., Berkeley Cal., May 1977